Standard OpenMP

OpenMP to jednolite środowisko dyrektyw zrównoleglających dla maszyn z pamięcią wspólną. Standard OpenMP obejmuje takie **dyrektywy** dla języków Fortran 90 oraz C/C++, a także towarzyszące im elementy – **bibliotekę procedur** oraz **zmienne środowiskowe**.

Linie dyrektyw OpenMP w C/C++ rozpoczynają się od ciągu znaków:

```
#pragma omp
```

zaś w Fortranie 90 od ciągu:

```
!$OMP
```

Składnia dyrektywy w C/C++:

```
#pragma omp <dyrektywa> <klauzule>
```

Uwaga! Kontynuacja linii dyrektywy - znak "\" na końcu.

Przykładowo dyrektywa zaznaczenia bloku równoległego w C/C++ ma postać:

```
#pragma omp parallel  <klauzule>
      <instrukcje>
```

a w Fortranie 90:

```
!$OMP PARALLEL <klauzule>
      <instrukcje>
!$OMP END PARALLEL
```

Zmienne środowiskowe dla OpenMP:

`OMP_NUM_THREADS, OMP_DYNAMIC, OMP_NESTED, OMP_SCHEDULE`

Ustalanie liczby wątków:

- statycznie – za pomocą zmiennej `OMP_NUM_THREADS` (np. `$ export OMP_NUM_THREADS=12`)
- dynamicznie jawnie – za pomocą funkcji `omp_set_num_threads(`*`liczba_watkow`*`)` ;
- dynamicznie automatycznie w czasie działania programu (ang. `runtime`; system sam decyduje ile ich będzie):
  - ustawiajac przed wywołaniem programu zmienną `OMP_DYNAMIC = TRUE`
  - lub wywołując w programie funkcję `omp_set_dynamic(`*`dynamiczne_watki`*`)` z parametrem *dynamiczne_watki* $\neq 0$

W celu dodatkowego rozszczepienia istniejących wątków (tzn. zagnieżdżonego zrównoleglenia) należy:

- ustawić zmienną `OMP_NESTED = TRUE`
- lub wywołać `omp_set_nested(`*`zagniezdzenie`*) dla *`zagniezdzenie`* $\neq 0$.

Zmienna `OMP_SCHEDULE` określa **przydział iteracji dla dyrektywy** `for`, gdy wykorzystywany jest sposób przydziału iteracji dla wątków w czasie wykonania. Zmienna może przyjąć wartości: `static, dynamic, guided` (wraz z opcjami).

Dyrektywy środowiska OpenMP:

1. Zaznaczenie bloku równoległego.

```
#pragma omp parallel  <klauzule>
      <instrukcje>
```

Dyrektywa ta definiuje blok równoległy. Wskazuje ona kompilatorowi, że to co będzie w bloku następującym po niej (zdefiniowanym, zgodnie ze składnią języka C, poprzez nawiasy klamrowe), ma być wykonywane przez wiele niezależnych wątków (procesorów na maszynie równoległej). Blok równoległy składa się z tzw. części redundantnej, w której wszystkie wątki robią to samo, oraz części podzielonej, w której (przede wszystkim dzięki zastosowaniu dyrektyw sterujących podziałem pracy) wykonują one różne części zadania zapisanego w programie sekwencyjnym. Wśród klauzul tej dyrektywy mogą wystąpić:

- `if(skalarne_wyrazenie_logiczne)`
  Warunek od którego zależy zrównoleglenie danego bloku; jeśli wyrażenie logiczne (podczas wykonywania programu) ma wartość `0 (FALSE)`, to instrukcje w tym bloku są wykonywane sekwencyjnie
- `private(lista_zmiennych_lokalnych)`
  Deklaracja zmiennych prywatnych dla każdego wątku z grupy (tzn. z danego bloku równoległego); domyślnie wszystkie zmienne zadeklarowane w bloku którego dotyczy ta dyrektywa (tzn. ze stosu wątku) są prywatne

- `shared(lista_zmiennych_wspólnych)`
  Deklaracja zmiennych współdzielonych, czyli wspólnych dla wszystkich wątków; domyślnie wszystkie zmienne widziane z danego bloku równoległego (np. globalne) - poza zmiennymi sterującymi pętli rozdzielanych za pomocą dyrektywy `omp for` - są współdzielone
- `default(shared|none)`
  Deklaracja, że wszystkie nie zadeklarowane zmienne widziane z danego bloku równoległego są albo wspólne albo, że wszystkie muszą być zadeklarowane jawnie. Jeśli tej deklaracji nie ma, przyjmuje się, że wszystkie te zmienne są wspólne (tak jak `default(shared)`)
- `firstprivate(lista)`
  Znaczenie to samo co deklaracji `private`, tylko zmiennym lokalnym z podanej listy we wszystkich wątkach nadaje się wartości początkowe takie, jakie miały przed wejściem do bloku równoległego (domyślnie wartości początkowe są nieustalone)
- `lastprivate(lista)`
  Znaczenie to samo co dyrektywy `private`, tylko po zakończeniu bloku równoległego, zmienne z podanej listy będą miały wartość taką, jak pod koniec wykonywania ostatniej instrukcji w wersji sekwencyjnej
- `reduction(operator: lista)`
  Obliczenie łącznych (uwzględniając wszystkie wątki) wartości operacji: `+`, `*`, `-`, `&`, $\wedge$, `|`, `&&`, `||` oraz `++`, `--` dla zmiennych z listy

- `copyin(lista)`
  Kopiowanie z wątku głównego do wszystkich wątków grupy tej samej wartości zmiennych z listy, które są wymienione również na liście dyrektywy `threadprivate`
- `num_threads(liczba)`
  Określenie liczby wątków dla bloku równoległego

Jeśli występuje kilka klauzul, są one oddzielone spacjami lub przecinkami.

Przykład: Podział gruboziarnisty obliczeń na równolegle wykonujące się wątki. Każdy wątek decyduje (bazując na numerze wątku), którą część globalnej tablicy `x` przetwarza:

```
#pragma omp parallel shared(x, npoints) private(iam, np, ipoints)
{
   iam = omp_get_thread_num();     /* identyfikacja watku      */
   np = omp_get_num_threads();     /* odczytanie liczby watkow */
   ipoints = npoints / np;
   subdomain(x, iam, ipoints);
}
```

2. Rozdzielenie (rozbicie) pętli.

`#pragma omp for` *<klauzule>*
<pętla for>

Jest to podstawowa dyrektywa dzieląca pracę między wątkami. Wskazuje ona, że poszczególne iteracje pętli `for`, w bloku tej instrukcji, mają być wykonywane niezależnie od siebie przez wiele wątków. Dotyczy ona tylko pętli iterowanych, ze zmiennymi sterującymi. Pętla `for` musi być w postaci kanonicznej, czyli:
`for (`*init-expr; var logical-op int-expr; incr-expr*`)`
Klauzule dla dyrektywy `for` to: `private, firstprivate, lastprivate, reduction, schedule, ordered, nowait`.
Na zakończenie pętli przyjmuje się, że jest bariera (można ją wyłączyć za pomocą klauzuli `nowait`).

Znaczenie 4 pierwszych klauzul - takie jak dla dyrektywy `parallel`; znaczenie pozostałych:

- `schedule(typ [,kwant])`
  Klauzula ta określa, jak iteracje pętli `for` mają być podzielone między wątki. Parametr `kwant` jest wyrażeniem, którego wartość określa kwanty przydziału iteracji do wątków. Parametr `typ` natomiast definiuje typ przydziału. Parametr `typ` może przyjąć następujące wartości:
  - `static`, czyli `schedule(static [,kwant])`
    Iteracje są dzielone na części wielkości `kwant`. Następnie są po kolei przydzielane do wątków, ewentualnie z zapętleniem, jeśli po jednym obiegu nie wszystkie części zostały przydzielone. W przypadku gdy parametr `kwant` nie jest podany, iteracje będą podzielone na tyle jednakowych, spójnych części, ile jest wątków, a te części następnie będą przydzielone po kolei do wszystkich wątków;

  - `dynamic`, czyli `schedule(dynamic [,kwant])`
    Iteracje są dzielone na części wielkości `kwant`. Jeśli jakiś wątek zakończy wcześniejszy kwant iteracji, dostaje nowy. Jeśli parametr `kwant` nie jest podany, przyjmuje się jego domyślną wartość równą `1`;
  - `guided`, czyli `schedule(guided [,kwant])`
    Iteracje są dzielone na części o wielkości malejącej wykładniczo, do wielkości równej `kwant` (domyślnie równej `1`). Wielkość kolejnych kawałków jest w przybliżeniu równa liczbie nieprzydzielonych iteracji dzielonej przez liczbę wątków. Następnie części są przydzielane dynamicznie do wątków.
  - `runtime`, czyli `schedule(runtime)`
    Decyzja o typie podziału jest odłożona do czasu uruchomienia programu i będzie przekazana za pomocą zmiennej środowiskowej `OMP_SCHEDULE`, np.:

```
$ export OMP_SCHEDULE="guided,4"
$ export OMP_SCHEDULE="dynamic"
```

- `ordered`
  Wewnątrz bloku instrukcji danej pętli `for` będzie blok instrukcji wykonywanych w takiej kolejności jak w programie sekwencyjnym (współpracuje z dyrektywą `#pragma omp ordered`)
- `nowait`
  wyłączenie bariery pod koniec pętli

Jeśli występuje kilka klauzul, są one oddzielone spacjami lub przecinkami.

Przykład: Zastosowanie dyrektywy - proste rozdzielenie pętli `for`:

```
#pragma omp for <klauzule>
        for (i = 1; i<n; i++) b[i] = a[i] + a[i-1];
```

Przykład: Statyczne rozdzielenie pętli `for` przy mnożeniu macierzy

```
#pragma omp parallel private(i, j, k) shared (a, b, c, dim) \
                     num_threads(4)
   #pragma omp for schedule(static)
   for (i=0; i < dim; i++) {
      for (j=0; j < dim; j++) {
         c[i][j] = 0;
         for (k=0; k < dim; k++) {
            c[i][j] += a[i][k] * b[k][j];
         }
      }
   }
```

Pętle `for` mogą występować w sekwencji lub być zagnieżdżone.

Pętle zagnieżdżone: zrównoleglenie pętli wewnętrznej nastąpi tylko wtedy, gdy znajdzie się ona w nowym bloku równoległym `parallel` i będzie wcześniej ustawiona zmienna środowiskowa `OMP_NESTED`.

Przykład: Jeśli jest wiele niezależnych pętli w obszarze równoległym, można wykorzystać klauzulę `nowait` do uniknięcia bariery na końcu dyrektywy `for`.

```
#pragma omp parallel
{
   #pragma omp for nowait
      for (i=1; i<n; i++)
         b[i] = (a[i] + a[i-1]) / 2.0;
   #pragma omp for nowait
      for (i=0; i<m; i++)
         y[i] = sqrt(z[i]);
}
```

Przykład: Prawidłowe działanie zależy niekiedy od wartości zmiennej podstawionej w ostatniej iteracji. Takie zmienne muszą być podane jako argumenty klauzuli `lastprivate`.

```
#pragma omp parallel
{
   #pragma omp for lastprivate(i)
      for (i=0; i<n-1; i++)
         a[i] = b[i] + b[i+1];
}
a[i]=b[i];
```

3. Sekwencyjne wykonywanie części instrukcji pętli

```
#pragma omp ordered
  {
    ⋮
  }
```

Zaznaczony blok instrukcji jest wykonywany w kolejności takiej jak w programie sekwencyjnym, a nie wynikającej z przebiegu realizacji poszczególnych wątków programu w czasie rzeczywistym. Dyrektywa ta może się znaleźć tylko wewnątrz bloku instrukcji for poprzedzonej dyrektywą rozdzielenia pętli (`#pragma omp for`), w której wystąpiła klauzula `ordered`. Każda iteracja pętli musi wtedy napotkać tę dyrektywę dokładnie raz (przy czym nie musi to być za każdym razem to samo jej wystąpienie).

Przykład: Uszeregowanie wyników obliczeń wykonanych równolegle za pomocą klauzuli `ordered`.

```
#pragma omp for ordered schedule(dynamic)
   for (i=lb; i<ub; i+=st)
      work(i);
void work(int k)
{
   :
   #pragma omp ordered
      printf("␣%d", k);
   :
}
```

4. Bloki (sekcje) wariantowe.

```
#pragma omp sections  <klauzule>
   {
   [#pragma omp section]
       <blok instrukcji>
   #pragma omp section
       <blok instrukcji>
   ⋮
   }
```

Dyrektywa `sections` służy do podziału między wątki bloku instrukcji innych niż pętle iterowane. Wśród klauzul mogą wystąpić `private, firstprivate, lastprivate, reduction, nowait`. Każdy blok jest poprzedzony dyrektywą `section` (dla bloku pierwszego jest ona opcjonalna). Domyślnie koniec bloku jest barierą dla wszystkich wątków, chyba że występuje klauzula `nowait`.

5. Lokalna jednowątkowość.

```
#pragma omp single  <klauzule>
    <blok instrukcji>
```

Wśród klauzul mogą wystąpić `private, firstprivate, lastprivate, copyprivate, nowait`. Dana część kodu jest wykonywana przez jeden wątek (zazwyczaj pierwszy, który do niej dotarł). Pozostałe wątki czekają przy barierze na końcu bloku, chyba że występuje klauzula `nowait`.

Klauzula `copyprivate` określa listę zmiennych (prywatnych), których wartości zostaną skopiowane z wątku, który wykonał sekcję `single` do lokalnych kopii należących do wszystkich pozostałych wątków. Klauzula ta występuje tylko w dyrektywie `single` i nie może być użyta łącznie z klauzulą `nowait`. Zmienne wymienione w tej klauzuli powinny być zmiennymi prywatnymi w aktualnym bloku równoległym i nie mogą pojawić się w klauzuli `private` ani `firstprivate` tej samej dyrektywy `single`.

6. Lokalna jednowątkowość ze wskazaniem wątku głównego.

```
#pragma omp master
    <blok instrukcji>
```

Dana część kodu wykonywana jest tylko przez wątek główny. Nie ma bariery ani na wejściu, ani na wyjściu z bloku. Dyrektywa nie przyjmuje żadnych klauzul.

Skrócone formy zaznaczenia bloków równoległych oraz rozdzielenia sterowania:

Dyrektywy te należy traktować tak, jak jednoczesne użycie dyrektywy `parallel` oraz danej dyrektywy rozdzielenia sterowania.

7. Skrócona forma zaznaczenia bloku równoległego oraz rozdzielenia (rozbicia) pętli

```
#pragma omp parallel for <klauzule>
        for (i = 1; i<n; i++)
           b[i] = a[i] + a[i-1];
```

Przykład: Skrócona forma użycia pętli z klauzulą `reduction`.

```
#pragma omp parallel for private(i) shared(x, y, n) \
                              reduction(+: a, b)
   for (i=0; i<n; i++) {
      a = a + x[i];
      b = b + y[i];
   }
```

8. Skrócona forma zaznaczenia bloku równoległego oraz bloków wariantowych

```
#pragma omp parallel sections <klauzule>
  {
  [#pragma omp section]
  ⋮
  #pragma omp section
  }
```

Synchronizacja danych oraz obliczeń:

9. Zamek (blokada, sekcja krytyczna)

```
#pragma omp critical[(nazwa)]
  <blok instrukcji>
```

Standardowy mechanizm ochrony sekcji krytycznej przed dostępem więcej niż jednego wątku. Opcjonalna `nazwa` określa ten sam zamek, który może być wykorzystany do blokowania dostępu do różnych sekcji krytycznych. Ten identyfikator nie może się już powtórzyć w programie.

10. Aktualizacja zmiennej po wykonaniu modyfikacji

```
#pragma omp atomic
```

Dyrektywa odnosi się do linii następującej bezpośrednio po niej. Może to być jedynie linia podstawienia spośród zestawu:

```
x operator = wyrazenie
++x , x++, --x , x--
```

gdzie `x` to zmienna skalarna, `wyrazenie` jest dowolnym wyrażeniem skalarnym nie odnoszącym się do `x` (nie jest ono objęte blokadą), `operator` jest z zestawu: `+, *, -, /, &, ∧, |, <<, >>` oraz `++, --`

Dyrektywa `atomic` ma sens uproszczonego zamka. Po wykonaniu operacji zmienna `x` jest modyfikowana w pamięci wspólnej, do której w tym czasie nie są dopuszczane inne wątki.

11. Natychmiastowa aktualizacja zmiennych we wszystkich wątkach grupy

```
#pragma omp flush [(lista)]
```

Dyrektywa wymusza zgodność wartości zmiennych wspólnych (z podanej listy) wszystkich wątków grupy. Jeżeli lista nie występuje, wszystkie zmienne współdzielone widziane z wątku są aktualizowane.

12. Bariera

```
#pragma omp barrier
```

Podstawowy mechanizm synchronizacji wątków. Bariera w OpenMP synchronizuje wszystkie wątki grupy.

13. Uczynienie ze zmiennych globalnych (pliku, przestrzeni nazw) zmiennych prywatnych wątków, z ich zachowaniem pomiędzy różnymi blokami równoległymi.

```
#pragma omp threadprivate (lista)
```

W niektórych sytuacjach jest to wygodniejsze niż wielokrotne klauzule `private` w każdym bloku równoległym. Jest to bardzo przydatna dyrektywa gdy się korzysta z cudzego oprogramowania, np. z bibliotek. Jej zastosowanie na początku programu wymusza by procedury te były MT-bezpieczne w całym programie (od ang. *MultiThreaded safe*), czyli żeby sobie ubocznie nie szkodziły. Dyrektywa ta może być również zastosowana do zmiennych lokalnych (bloków) statycznych.

Funkcje biblioteczne: Oprócz dyrektyw i zmiennych środowiskowych, standard OpenMP obejmuje jeszcze bibliotekę procedur. Wszystkich procedur w bibliotece jest obecnie około dwudziestu. Poza wymienionymi już: `omp_set_num_threads(int)`, `omp_set_dynamic(int)`, `omp_set_nested`, warto wspomnieć o:

- 10 funkcji dotyczy identyfikacji i zmiany środowiska wykonywania obliczeń, najważniejsze:

  `int omp_get_num_threads(void)` – funkcja zwracająca liczbę wątków w danej części kodu;

  `int omp_get_thread_num(void)` – funkcja zwracająca numer bieżącego wątku (wątki są numerowane od numeru `0`, przypisanego wątkowi głównemu);

  `int omp_get_num_procs(void)` – funkcja zwracająca liczbę procesorów w danej maszynie;

  `int omp_in_parallel(void)` – predykat, funkcja informująca, czy została wywołana z bloku równoległego, którego instrukcje są rzeczywiście wykonywane równolegle;

- procedurach związanych z dynamicznym powoływaniem i usuwaniem zamków (ang. *lock routines*): `omp_init_lock`, `omp_set_lock`, `omp_destroy_lock`, `omp_unset_lock`, `omp_test_lock`. Funkcje mogą być wykorzystane np. w tych miejscach programu, gdzie nie jest jasne $a\ priori$, czy modyfikacja elementów powinna znaleźć się w sekcji krytycznej. Parametrem wywołania wszystkich tych funkcji jest `omp_lock_t *lock`.

- procedurach związanych jest z dynamicznym powoływaniem i usuwaniem zamków zagnieżdżonych: `omp_init_nest_lock`, `omp_set_nest_lock`, `omp_destroy_nest_lock`, `omp_unset_nest_lock`, `omp_test_nest_lock`. Parametrem wywołania wszystkich tych funkcji jest `omp_nest_lock_t *lock`.

- funkcjach związanych z pomiarami czasu (ang. *timing routines*), w tym:

  `double omp_get_wtime(void)` - funkcja odczytująca aktualny czas (ang. *wall clock time*) w sekundach od pewnej chwili w przeszłości. Funkcja ta jest bardzo przydatna np. do obliczania czasu wykonania programu oraz współczynnika przyśpieszenia.

  `double omp_get_wtick(void)` - funkcja odczytująca okres zegara maszyny w sekundach, czyli ile czasu mija między dwoma "tyknięciami". Odwrotność tej wielkości poda nam częstotliwość zegara procesora.

Uwaga! Jeśli się korzysta z biblioteki procedur OpenMP należy na początku programu umieścić linię:

```
#include <omp.h>
```

Przykład: Pobranie informacji o liczbie wątków BŁĄD!.

```
np = omp_get_num_threads();
#pragma omp parallel for schedule(static)
   for (i=0; i<np; i++) {
      work(i);
   }
```

Pobranie informacji o liczbie wątków DOBRZE.

```
#pragma omp parallel private(i)
{
      i = omp_get_num_threads();
      work(i);
}
```

Przykład 1

Obliczanie sumy n liczb: `suma=a[0]+a[1]+...+a[n-1]`

Pętla jest realizowana równolegle na `p` procesorach. Kolejne wątki będą wykonywały obliczenia dla wartości `i`=1,...,n/p; `i`=n/p+1,...,2n/p; itd.

Zmienne:

`n` i `a` – dzielone (`shared`) – wszystkie procesory mają do nich dostęp

`i` – prywatna (`private`) – każdy wątek ma swoją kopię zmiennej o nazwie `i`

`sum` – zmienna podlegająca redukcji `(reduction+:sum);` oznacza to, że podczas obliczeń tworzona jest lokalna kopia tej zmiennej na każdym procesorze. Na koniec pętli wszystkie lokalne `sum` są dodawane (został podany operator "+").

```
float array_sum(float a[], int n) {
   int i;
   float sum = 0.0;
   #pragma omp parallel
   {
      #pragma omp for shared(a,n) private(i) reduction(+:sum)
      {
         for (i=0; i<n; i++)
            sum += a[i];
      }
   }
   return(sum);
}
```

□

Przykład 2

Odczytanie częstotliwości zegara maszyny

Oto przebieg pewnej sesji na maszynie Sun Fire V440 z czterema procesorami:

```
mion<XXXXXXXX>(634)~$ mpstat
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0  275   1  840   162   27  638    8   52   75    0  3470    7   5   0  88
  1  106   2  389   163    5  460    6   40   92    0  2673    4   2   0  93
  2   80   3  378   247  168  274    5   29   47    0  2893    4   2   0  94
  3  541   1  326   382  185  678   11   51   42    0  2962    6   4   0  90
mion<XXXXXXXX>(635)~$ cat ompTestclock.c
```

```
/*                                                    */
/*  Program ompTestclock.c                            */
/*  Odczytywanie czestotliwosci zegara procesorow     */
/*                                                    */
#include <stdio.h>
#include <omp.h>
main(){
   double timerd, czestotl;
#pragma omp parallel private(timerd, czestotl)
{
   timerd=omp_get_wtick();
   czestotl=1./(1e9*timerd);
   printf("wtick=%20.20f\n", timerd);
   printf("czestotliwosc␣zegara=%f␣GHz\n", czestotl);
}
}
```

```
mion<XXXXXXXX>(636)~$ cc -xopenmp ompTestclock.c
cc: Warning: Optimizer level changed from 0 to 3 to support parallelized code.
mion<XXXXXXXX>(637)~$ export OMP_NUM_THREADS=1
mion<XXXXXXXX>(638)~$ ./a.out
wtick=0.00000000062774639046
czestotliwosc zegara=1.593000 GHz
mion<XXXXXXXX>(639)~$ export OMP_NUM_THREADS=4
mion<XXXXXXXX>(640)~$ ./a.out
wtick=0.00000000062774639046
czestotliwosc zegara=1.593000 GHz
wtick=0.00000000062774639046
czestotliwosc zegara=1.593000 GHz
wtick=0.00000000062774639046
czestotliwosc zegara=1.593000 GHz
wtick=0.00000000062774639046
czestotliwosc zegara=1.593000 GHz
mion<XXXXXXXX>(641)~$
```

□

Przykład 3

Separowalna funkcja dwóch wektorów.

Obliczanie pewnej funkcji skalarnej dwóch wektorów 200-elementowych – z zapisem do pliku 'wyniki.dat' iteracji pętli wykonywanych przez kolejne procesory wchodzące do sekcji krytycznej (nie są to na ogół kolejne iteracje $0, 2, 3, ..., 199$). Dzięki temu można zobaczyć, jak jest rozwijana pętla liczenia kolejnych składników tej funkcji (zależnych od iloczynów kolejnych elementów tych wektorów) i że zawsze wynik jest taki sam. Jest to zasługa zamka chroniącego zmienną `sum`), mimo różnej kolejności wykonywania iteracji.

```
/*                                                              */
/*   ompTstIlsk1.c   - pierwsza wersja, bardziej czytelna       */
/*                     (z deklaracjami private/shared)          */
#include <stdio.h>
#include <math.h>
#define lba_iter 200
main(){
   float a[lba_iter], b[lba_iter];
   int ind_lok[lba_iter], tab_it_wyk[2*lba_iter];
   double sum;
   double xsum;
   int it_wpis;
   int i, iter_no, j, k;
   FILE *fp, *fopen();
   for(i=0;i<lba_iter;i++){
         a[i]=i;b[i]=i+2;
}
   it_wpis=0.; sum=0.;
#pragma omp parallel private(xsum,iter_no,ind_lok, k, j) \
                     shared(sum,it_wpis, tab_it_wyk)
{
   double coslicz, sinlicz, arglicz;
```

```
   xsum = 0.0; iter_no=0;
#pragma omp for
   for(i=0;i<lba_iter;i++){
         sinlicz = sin((double)a[i]*b[i]);
         arglicz=0;
         for(k=1; k< 5000;k++){
            arglicz=arglicz+k*sinlicz;
            coslicz=cos(arglicz*arglicz);
         }
         xsum = xsum+coslicz;
         ind_lok[iter_no] = i;
         iter_no=iter_no+1;
   }
#pragma omp critical
   {
      for(j=0; j< iter_no; j++){
          tab_it_wyk[it_wpis+j] = ind_lok[j];
      }
      it_wpis = it_wpis + iter_no;
      sum = sum + xsum;
   }
}
   fp=fopen("wyniki.dat","w");
   for(i=0;i<lba_iter;i++){
       fprintf(fp,"%d␣",tab_it_wyk[i]);
       if(i>0 && (i+1)%10==0)fprintf(fp,"\n");
   }
   fprintf(fp,"\n%lf\n", sum);
   printf("Suma=%lf\n", sum);
}
```

A oto przebieg pewnej sesji na maszynie Sun Fire V440 z czterema procesorami:

```
mion<XXXXXXXX>(283)~$ cat ompTstIlsk2.c

/*                                                              */
/*   opmTstIlsk2.c   - druga wersja, mniej czytelna            */
/*                     (bez deklaracji private/shared)         */

#include <stdio.h>
#include <math.h>
#define lba_iter 200
main(){
   float a[lba_iter], b[lba_iter];
   int tab_it_wyk[2*lba_iter];
   int it_wpis;
   double sum;
   int  i, j;
   FILE *fp, *fopen();
   for(i=0;i<lba_iter;i++){
         a[i]=i;b[i]=i+2;
}
   it_wpis=0.; sum=0.;
#pragma omp parallel
{
   double coslicz, sinlicz, arglicz;
   double  xsum;
   int ind_lok[lba_iter];
   int  iter_no, k;
   xsum = 0.0; iter_no=0;
#pragma omp for
   for(i=0;i<lba_iter;i++){
         sinlicz = sin((double)a[i]*b[i]);
         arglicz=0;
```

```
         for(k=1; k< 5000;k++){
            arglicz=arglicz+k*sinlicz;
            coslicz=cos(arglicz*arglicz);
         }
         xsum = xsum+coslicz;
         ind_lok[iter_no] = i;
         iter_no=iter_no+1;
   }
#pragma omp critical
   {
      for(j=0; j< iter_no; j++){
          tab_it_wyk[it_wpis+j] = ind_lok[j];
      }
      it_wpis = it_wpis + iter_no;
      sum = sum + xsum;
   }
}
   fp=fopen("wyniki.dat","w");
   for(i=0;i<lba_iter;i++){
       fprintf(fp,"%d ",tab_it_wyk[i]);
       if(i>0 && (i+1)%10==0)fprintf(fp,"\n");
   }
   fprintf(fp,"\n%lf\n", sum);
   printf("Suma=%lf\n", sum);
}
```

```
mion<XXXXXXXX>(284)~$ diff ompTstIlsk1.c ompTstIlsk2.c
1,3c1,4
< /*                                                          */
< /*   ompTstIlsk1.c   - pierwsza wersja, bardziej czytelna */
< /*                     (z deklaracjami private/shared)    */
---
> /*                                                            */
> /*   opmTstIlsk2.c   - druga wersja, mniej czytelna           */
> /*                     (bez deklaracji private/shared)        */
>
9,11c10
<    int ind_lok[lba_iter], tab_it_wyk[2*lba_iter];
<    double sum;
<    double xsum;
---
>    int tab_it_wyk[2*lba_iter];
13c12,13
<    int i, iter_no, j, k;
---
>    double sum;
>    int  i, j;
19,20c19
< #pragma omp parallel private(xsum,iter_no,ind_lok, k, j) \
<                         shared(sum,it_wpis, tab_it_wyk)
---
> #pragma omp parallel
22a22,24
>    double  xsum;
>    int ind_lok[lba_iter];
>    int  iter_no, k;
```

```
mion<XXXXXXXX>(285)~$ cc -xopenmp ompTstIlsk2.c -lm
cc: Warning: Optimizer level changed from 0 to 3 to support parallelized code.
mion<XXXXXXXX>(286)~$ export OMP_NUM_THREADS=1
mion<XXXXXXXX>(287)~$ timex ./a.out
Suma=12.590631

real        0.92
user        0.91
sys         0.00

mion<XXXXXXXX>(288)~$ export OMP_NUM_THREADS=2
mion<XXXXXXXX>(289)~$ timex ./a.out
Suma=12.590631

real        0.46
user        0.91
sys         0.00

mion<XXXXXXXX>(290)~$ export OMP_NUM_THREADS=4
mion<XXXXXXXX>(291)~$ timex ./a.out
Suma=12.590631

real        0.24
user        0.92
sys         0.00
```

```
mion<XXXXXXXX>(292)~$ cat wyniki.dat
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169
170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189
190 191 192 193 194 195 196 197 198 199
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
100 101 102 103 104 105 106 107 108 109
110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129
130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149

12.590631
mion<XXXXXXXX>(293)~$
```

□

Linux, kompilator gcc od wersji 4.1.1

```
gcc  -fopenmp  [inne_opcje_kompilatora]  nazwa_pliku
```

Środowisko Omni OpenMP

Środowisko Omni OpenMP do programowania równoległego na maszynach wieloprocesorowych ze wspólną pamięcią (SMP) oraz klastrach (SMP/PC). Jest to zbiór programów i bibliotek napisanych w językach C i Java. Kompilatory wchodzące w skład Omni OpenMP umożliwiają translację programów napisanych w C oraz Fortranie z pragmami C na kod języka C, który następnie jest kompilowany przez kompilatory właściwe dla danego systemu operacyjnego. Kod wynikowy jest konsolidowany z bibliotekami czasu wykonania (runtime) Omni OpenMP. Środowisko Omni OpenMP działa na wielu platformach sprzętowych i systemowych m.in.:

- Sun Solaris (SPARC i x86) z wątkami Solaris oraz POSIX.
- Linux (Redhat, x86, Alpha SMP) z wątkami Linuksa.
- IRIX 6.5 (Origin 2000).
- IBM AIX z wątkami POSIX.
- HPUX z wątkami POSIX.

Uruchomienie kompilatora Omni OpenMP C:

```
omcc  [opcje_drivera]  [opcje_kompilatora]  nazwa_pliku
```